Introducing Energy Efficiency into SQALE

Introducing Energy Efficiency into SQALE

Luca Ardito, Giuseppe Procaccianti, Antonio Vetro', Maurizio Morisio
Dipartimento di Automatica ed Informatica
Politecnico di Torino
Torino, Italy
E-mail: name.surname@polito.it

***Abstract*—Energy Efficiency is becoming a key factor in software development, given the sharp growth of IT systems and their impact on worldwide energy consumption. We do believe that a quality process infrastructure should be able to consider the Energy Efficiency of a system since its early development: for this reason we propose to introduce Energy Efficiency into the existing quality models. We selected the SQALE model and we tailored it inserting Energy Efficiency as a sub-characteristic of efficiency. We also propose a set of six source code specific requirements for the Java language starting from guidelines currently suggested in the literature. We experienced two major challenges: the identification of measurable, automatically detectable requirements, and the lack of empirical validation on the guidelines currently present in the literature and in the industrial state of the practice as well. We describe an experiment plan to validate the six requirements and evaluate the impact of their violation on Energy Efficiency, which has been partially proved by preliminary results on C code. Having Energy Efficiency in a quality model and well verified code requirements to measure it, will enable a quality process that precisely assesses and monitors the impact of software on energy consumption.**

Keywords-Energy Efficiency; energy-aware software; SQALE

I. INTRODUCTION

The rapid growth and significant development of Information Technology (IT) systems has started to cause an increase of worldwide energy consumption [1]. This issue moved technology producers, information systems managers, and researchers to deal with energy consumption reduction [2]. For this reason, research has increasingly focused on improving the Energy Efficiency of hardware, but the literature still lacks in quantifying accurately the energy impact of software. Software does not consume energy directly, however it has a direct influence on the energy consumption of the hardware underneath. As a matter of fact applications and operating systems indicate how the information is processed and, consequently, drive the hardware behaviour. Considering each IT device, it has its own theoretical energy consumption, which can range from 0, when it is turned off, to x if all its internal components are used simultaneously. Through the management of each part there is a variation Δx of its consumption that is between 0 and x. The management of system components can be done either in hardware or software. Previous work [3] suggested

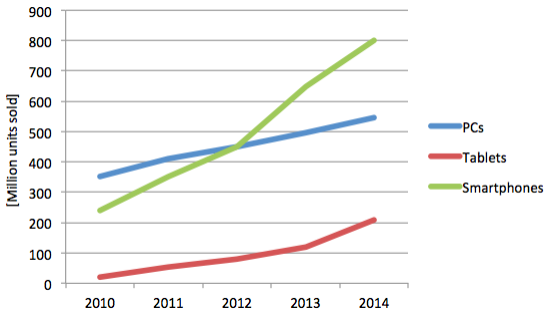

Figure 1. IT devices sale forecasts

that software can reach up to 10% of the total system power (measured as the difference between an idle activity, used as a baseline, and the most power-consuming scenario). This figures ought to be taken into account especially when considering mobile environments and data centers. Mobile handsets sales are increasing sharply [4] (see Fig. 1) and this class of devices have to deal with battery-related issues, so energy savings can impact significantly on the device autonomy. On the other hand, small energy reductions in data centers can result in big energy savings: for example, just in 2009, data centers consumed about 330 TWh [5].

Having regard to the influence of software in energy consumption, it is necessary to quantify the Energy Efficiency of source code. For this reason, we envision a software quality model that includes Energy Efficiency in order to take it into account as a key aspect during the software development and utilization. Having in mind this scenario, we suggest the usage of the Software Quality Assessment based on Lifecycle Expectations (SQALE) model [6] to include Energy Efficiency as a measurable quality attribute. This paper is organized as follows. Section II describes SQALE methodology. Section III discusses the adaptation of SQALE quality model to include Energy Efficiency. Section IV introduces a software framework for Requirements Empirical Validation. Section V presents our conclusions and future work.

II. SQALE

SQALE [6] is a methodology to support the evaluation of the software quality. It is applicable to any software artifact (such as code, UML models, documentation, and so on),

 ISBN: 978-1-61208-259-2

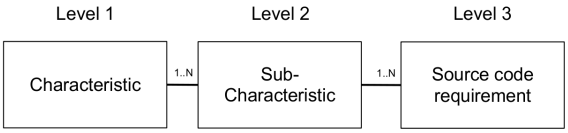

Figure 2. Hierarchical quality model structure

however the main focus is on source code, whose quality is perceived as a non functional requirement. The goal of using SQALE is to quantitatively assess the distance between the code current quality and its expected quality objective. To achieve that, the following main concepts are introduced:

1) A quality model
2) An analysis model
3) Indices and Indicators

A. Quality model

The quality model proposed by SQALE is an orthogonal quality model derived from the ISO/IEC 9126 [7] (revised by the ISO/IEC 25010 [8]) . It is organised in three hierarchical levels, which are represented in Fig. 2. The first level is composed of characteristics that are based on the theoretical lifecycle of a source file and are from the ISO 9126 standard. They depend on the code internal properties and directly impact the typical activities of a software application's lifecycle. Characteristics are listed in the order they appear in a typical software application lifecycle: Testability, Reliability, Changeability, Efficiency, Security, Maintainability, Portability, Reusability.

The second level is composed of sub-characteristics, based on sub activities and requirements domain. There are two types of sub-characteristics: those corresponding to lifecycle activities (e.g., unit test, integration test), and those resulting from taxonomies in terms of good and bad practices relating to the software's architecture and coding. A sub-characteristic is attached to only one characteristic, the first in the chronology of the characteristics (to preserve orthogonality). The third level is composed of requirements that relate to the source code's internal attributes. These requirements usually depend on the software's context and language, and they are also attached to the lowest possible level, i.e. in relation to the first quality characteristic to which it chronologically contributes. In this way orthogonality is preserved also at the bottom level. Requirements relate to the artifacts that compose the software's source code, e.g. software applications, components, files, classes, and so forth. TABLE I is excerpt from the SQALE standard [6] and it contains examples of how requirements in the Java language are inserted in the structure of characteristics and sub-characteristics. Fig. 3 represents graphically the hierarchy.

B. Analysis model

The SQALE Analysis Model contains the rules to normalize and control measures relating to the code. For each violated source code requirement, a remediation cost (a work unit, a monetary unit, or a time unit) is associated to make the code conformant to the quality objective. For instance, looking at TABLE I and Fig. 3, the remediation cost for the requirement "Coupling between objects (CBO) < 7" is the cost to decrease the coupling from its current value X to 7. The remediation cost might not be constant but expressed by a remediation function. For example, reducing the coupling from 13 to 7 (-6) might cost more than twofold effort of reducing it from 10 to 7 (-3). The total refactoring cost for a sub-characteristic is the sum of the remediation cost of each requirement violation.

TABLE I
EXAMPLE OF SQALE MODEL FOR JAVA LANGUAGE, FROM [6]

Characteristic	**Sub-characteristic**	**Generic Requirement Description**
Maintainability	Understandability	File comment ratio $> 35\%$
Maintainability	Readability	File size (LOC) < 1000
Maintainability	Readability	No commented-out code
Efficiency	RAM related efficiency	Class depth of inheritence (DIT) < 8
Efficiency	RAM related efficiency	No unused variables, parameter or constant in code
Changeability	Logic related changeability	If, else, for, while structures are bound by scope
Reliability	Fault tolerance	Switch statements must have a default condition
Reliability	Data related reliability	No use of unitialized variables
Testability	Integration level testability	Coupling between objects (CBO) < 7
Testability	Unit Testing testability	No duplicate part over 100 token
Testability	Unit Testing testability	Number of parameters in a module call (NOP) < 6

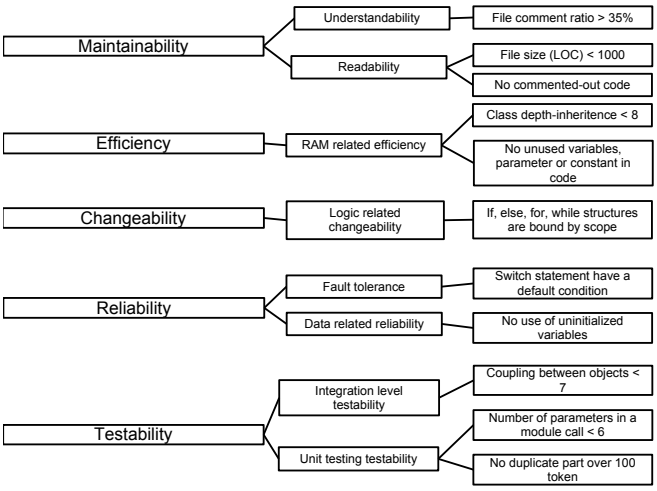

Figure 3. Hierarchical representation of the model described in TABLE I

C. Indices and Indicators

All of the SQALE indices represent the costs related to a given characteristic, estimated by adding up all of the remediation costs of the requirement violations of the connected sub-characteristics. For instance, in the example of TABLE I, the SQALE Testability Index STI is the sum of the remediation costs of all violated requirements related to Integration level testability and Unit testing testability, i.e. "No duplicate part over 100 token", "Number of parameters in a module call < 6", "Coupling between objects < 7". The sum of all indexes is the Software Quality Index (SQI). It is also possible to obtain consolidated indices in the following way: the consolidated index of a given characteristic is equal to the sum of all the indices of the previous characteristics. For instance, the SQALE Consolidated Reliability Index (SCRI) is equal to STI + SRI, i.e. the sum of Testability and Reliability indexes. Moreover, a density index has to be associated with each absolute index dividing it by a measure representing the size of the artifact (lines of code, complexity, etc.). Finally the SQALE Method defines several synthesised indicators to summarize the overall quality status of the application: since indicators are out of the scope of the current work, we point the reader to the SQALE document to more detailed information.

III. Tailoring SQALE quality model to include Energy efficiency

We propose to tailor the SQALE model to include Energy Efficiency. As stated in the introduction, including Energy Efficiency in a quality model is an important step towards a measurable, repeatable and objective way to evaluate and improve the Energy Efficiency of a given application. We suggest to introduce Energy Efficiency as a sub-characteristic related to the main characteristic "Efficiency". It cannot be a characteristic itself, because it is not an activity in the typical software lifecycle, but it is a sub-characteristic of second type (i.e., "a recognised taxonomy in terms of good and bad practices relating to the software's architecture and coding"). The next step is to identify appropriate code requirements that can be applied to evaluate the Energy Efficiency of a software product. For this purpose, in this section we propose a list of guidelines, derived from the literature [9] [10] [11], provided as solutions to developers in order to produce energy-efficient software. From these guidelines, we extract a set of proper requirements to be included in the SQALE Quality Model. Most of the guidelines suggested in the literature are not strictly code-related, but rather recommending general programming techniques. As a consequence, we selected from the original list those requirements that can be traced to actual code structures. These are listed in TABLE II.

Starting from the selected guidelines, we express the set of requirements for evaluating software Energy Efficiency.

TABLE II
Guidelines that can be translated into SQALE requirements

Nr.	*Guideline*	*Explanation*
GD1	Decrease algorithm complexity	Despite different algorithms can complete the same task, the way the task is performed can be totally different. Reducing the algorithm complexity can lead to save energy.
GD2	Use Event-Based programming	Event based programming avoids a waste of resources involved in doing unnecessary operations. If polling cannot be avoided, it is advised to select a fair time interval.
GD3	Batch I/O	Buffering I/O operations increases Energy Efficiency; the OS can power down I/O devices when not used.
GD4	Reduce data redundancy	Storage and transportation of redundant data impacts Energy Efficiency
GD5	Reduce memory leaks	With memory leaks the application can stall or crash. This unpredictable behavior can alter the energy consumption and, more generally, they must always be avoided.

These requirements, as specified in the SQALE Model Definition Document, [6], must be:

- Atomic
- Unambiguous
- Non-redundant
- Justifiable
- Acceptable
- Implementable
- Not in contradiction with any other requirement
- Verifiable

Our approach, in coherence with the SQALE methodology, is based upon translating the guidelines into code patterns automatically detectable with static analysis tools. We propose an estimate, basing upon the presence of particular implementations that may cause energy waste. Since requirements are meant by SQALE to be language-dependant, we use the Java language as a reference in this paper. TABLE III contains the requirements identified and mapped to the guidelines they derive from. This is not to be intended as an exhaustive list but a first step towards source code Energy Efficiency quantification.

A. RQ1: Halstead's Effort $< K$

Halstead's Effort [12] is a technique for describing the structural properties of algorithms. This metric has been selected because it gives an estimation of algorithm complexity, which is language-dependant, but not implementation-dependant as other metrics commonly used in this field (such as McCabe's Complexity [13]). K is a parameter to be defined according to specific application domains and project characteristics.

 ISBN: 978-1-61208-259-2

TABLE III
REQUIREMENTS FOR ENERGY EFFICIENCY

Nr.	*Guideline*	*Nr.*	*Requirement*
GD1	Decrease algorithm complexity	RQ1	Halstead's Effort$< K$
GD2	Use Event-Based programming	RQ2	Nr. of polling cycles=0
GD3	Batch I/O	RQ3	Nr. of FileInputStream.read() method calls = 0 [8]
GD4	Reduce data redundancy	RQ4	Nr. of unused variables = 0
GD5	Reduce memory leaks	RQ5.1	Nr. of Dead Store issues per class = 0
		RQ5.2	Nr. of String|Boolean|Integer|Double constructor = 0

B. RQ2: Nr. of polling cycles = 0

To date and up to our knowledge, no static analysis tool is able to detect polling cycle, because polling structures can be implemented in various ways. However, we decided to keep this requirement and to devote further work to find a relevant metric to detect polling.

C. RQ3: Nr. of FileInputStream.read() method calls = 0

This requirement derives from a particular issue regarding the FileInputStream.read() method, that triggers a direct call to the underlying OS. If inserted into a cycle, it will realize an inefficient I/O policy. The use of a BufferedReader greatly improves performance and supposedly Energy Efficiency of the operation [14]. For example, the code shown in Listing 1 continuously calls the read() method of a FileInputStream object, thus triggering a large number of RPC calls to the Operating System.

```
FileInputStream fis = new FileInputStream(filename]);
int cnt = 0;
int b;
while ((b = fis.read()) != −1)
{
        if (b == '\n')
                cnt++;
}
fis.close();
```

Listing 1. Example of inefficient I/O policy

The code shown in Listing 2 makes use of a BufferedInputStream, which reads larger chunks of data than the FileInputStream. This greatly reduces Remote Procedure Calls, which improves Energy Efficiency by allowing the Operative System (OS) to turn off the I/O device when not needed.

D. RQ4: Nr. of unused variables = 0

The code in Listing 3 shows an example of Unused Field issue: the AClass contains a private field named "b", which is never used (the class does not provide a get() method for that field).

```
FileInputStream fis = new FileInputStream(filename);
BufferedInputStream bis = new BufferedInputStream(fis);
int cnt = 0;
int b;
while ((b = bis.read()) != −1)
{
    if (b == '\n')
            cnt++;
}
bis.close();
```

Listing 2. Example of efficient I/O policy

```
private class AClass
{
        int a;
        int b;

        public int getA(){return a;}
}
```

Listing 3. Example of Unused Field

An optimization of this code would be providing a get() method for the "b" field, or removing the field if unnecessary.

E. RQ5.1: Nr. of Dead Store issues = 0

The code shown in Listing 4 contains a Dead Store issue, which means assigning a value to a local variable which is not read by any subsequent instruction.

```
public int DeadLocalStore(int x)
{
        int constant_a = x;
        constant_a = 3

        return constant_a + x;
}
```

Listing 4. Example of Dead Local Store

In the code above, x is stored to *constant_a* but it is overwritten in the subsequent code line. A more efficient code is shown in Listing 5.

```
public int noDeadLocalStore(int x)
{
        int constant_a = 3;
        return constant_a + x;
}
```

Listing 5. Example of refactored Dead Local Store

The value of x is no longer stored to *constant_a* and then replaced.

The requirements specified above are derived from the guidelines [11] [10] of good programming practices provided in the literature. However, it is worth mentioning that

such guidelines, despite being intuitive and acknowledged as effective by software industry specialists [1], did not receive any empirical validation. For this reason, and in order to make the choice of the above specified requirements *justifiable*, an empirical validation that quantitatively assess their impact on Energy Efficiency is needed.

The last steps in the introduction of Energy Efficiency into SQALE are: tailoring the analysis model, tailoring the indices and the indicators. These steps are more straightforward then the previous one. Regarding the analysis model, a remediation function for each requirement violation ought to be defined in order to obtain the remediation cost. An example of remediation function for the requirement

$$\text{number of dead stores} > 0$$

could be:

$$10 + 2x$$

where x is the number of deadstores, 10 is the cost (in time units) of running a static analysis tool to detect them and 2 is the estimated time cost to review and refactor each dead store. We plan to estimate the remediation cost of each requirement violation through controlled experiments (e.g. observing the time required by subjects for a refactoring action) and questionnaires (i.e. asking directly to practicioners for estimations of refactoring actions). Being Energy Efficiency a sub-characteristic of efficiency, the sum of the remediation costs of all its source requirements will be added to the total cost of the other efficiency sub-characteristics, obtaining the SQALE Efficiency Index. Finally, the indicators do not need tailoring because they are at the highest level of the quality model.

IV. Requirements Empirical Validation: Experiment planning and Preliminary Results

As said in the previous section, the guidelines we propose are not supported by an empirical evidence regarding their impact on Energy Efficiency. We plan an experiment to test whether the requirements identified in TABLE II have a measurable impact on energy consumption. We use as example the RQ 5.1 (dead stores) to explain the experiment framework. In our experimentation, we set up two source code fragments: one containing a dead store to an Integer variable and the corresponding refactored version, functionally identical but without the dead store. The instrumentation required for our experiment is a system to execute the code, meanwhile logging power consumption data through a Data Acquisition Board. On the software level, the infrastructure implementing the experiment is inspired by the JUnit2 [15] framework for automated software It consists of an abstract class, Experiment, that can be extended by concrete experimental classes. Each experimental class must provide two methods performWithViolation() and performWithoutViolation() that contain respectively the code including the violation and with the violation refactored out. In addition the method setUp() may be optionally redefined to prepare for the execution. For instance, the experimental class for the integer dead store explained in the previous section is shown in Listing 6.

```
public class DLS_DEAD_LOCAL_STORE extends Experiment
{
        public int performWithViolation()
        {
                int constant_a = x;
                constant_a = 3;
                return constant_a + x;
        }

        public int performWithoutViolation()
        {
                int constant_a = 3;
                return constant_a + x;
        }
}
```

Listing 6. Dead Local Store experimental class

The power consumption of the methods performWithViolation() and performWithoutViolation() are expected to be very small. Unfortunately the standard measurement methods are not able to record precisely power at order of magnitude below microWatts. For this reason, the execution of each method is repeated consecutively a very high number of times (e.g. 1 million) to consume a measurable amount of power. We assume that each execution of the measured methods is independent on each other. This is true if no attribute is used except those initialized in the setUp() method.

Moreover, the framework provides the method: perform(int nSamples , long nIter). It takes as parameters two integers: the number of measurement samples to be generated (nSamples, set to 100 by default), and the number of iterations of the perform methods (nIter, set to 1 million by default). At the end of the experiment we will have nSamples samples, each of them representing the execution times of nIter iterations of both perform methods. We also plan to have a batch of different runs of the basic experiment carried on at different random times during the day to compensate the possible confounding effect of periodical tasks performed by the operating system. Finally, having the power consumption data of the two methods, it is possible to compare them and assess the possible impact of the requirement violation on Energy Efficiency.

To date, we have finished the instrumentation of a Desktop machine where to run the experiment, by means of an electrical power meter [16], through which we will log power consumption data during the execution of software specifically written for our experimental purposes. We previously conducted a similar experiment on an embedded system with a integer dead store implemented in the C

 ISBN: 978-1-61208-259-2

language. Our results show the impact related to the dead store was in the order of 20-40 picowatts per instruction. Such a small power saving could be higher if code patterns are executed thousands or millions of time, as it might happen in loops. Moreover, considering devices running on batteries, dead store have a negative impact on battery consumption.

V. Conclusions and Future Work

Energy efficiency is becoming a key factor in software development, given the ubiquity of software in everyday life and its hardware-related power consumption. Moreover, in devices running on batteries, efficient energy consumption is a key aspect. For this reason we propose to introduce Energy Efficiency into the existing quality models. We selected SQALE, whose quality model is derived from the ISO/IEC 9126 and it is strictly related to the software lifecycle activities. We tailor SQALE inserting Energy Efficiency as a sub-characteristics of efficiency, and we propose a set of specific requirements for the Java language starting from guidelines currently developed in the literature. The requirements identified are:

- Halstead's Effort $< K$
- Nr. of polling cycles = 0
- Nr. of FileInputStream.read() method calls = 0
- Nr. of dead store issues per class = 0
- Nr. of unread variables = 0
- Nr. of String|Boolean|Integer|Double constructor = 0

We identified two major challenges in requirements elicitation:

1) the translation of the guidelines in measurable requirements, whose violations are automatically identifiable by tools;
2) the validation of the negative impact of the requirements violation on energy consumption.

We are planning an experiment to empirically verify the impact of requirements on Energy Efficiency and we presented the results of a preliminary work for an integer dead store implemented in the C language, where we verified that it actually causes an increase of power consumption per instruction. Future work will be devoted to execute the experiment to empirically validate the requirements, estimating both their negative impact on power consumption and the related remediation costs. We will also investigate whether other requirements are eligible to be included in the quality model under Energy Efficiency sub-characteristic.

References

[1] The Climate Group, "Smart 2020: Enabling the low carbon economy in the information age," GeSi, Tech. Rep., 2008.

[2] A. Berl et al., "Energy-efficient cloud computing," The Computer Journal, vol. 53, no. 7, pp. 1045–1051, 2010. [Online]. Available: http://comjnl.oxfordjournals.org/content/53/7/1045.abstract

[3] G. Procaccianti, A. Vetro, L. Ardito, and M. Morisio, "Profiling power consumption on desktop computer systems," in Information and Communication on Technology for the Fight against Global Warming, ser. Lecture Notes in Computer Science, D. Kranzlmller and A. Toja, Eds. Springer Berlin / Heidelberg, 2011, vol. 6868, pp. 110–123.

[4] "Global PC, Tablet, Smartphone Sales," 2010, last accessed on January 8, 2013. [Online]. Available: http://en.wiser.org/article/61670dc8dd5c5a3adba285f39bbf8145

[5] J. Scaramella, "Worldwide Server Energy Expense 2009–2013 Forecast," International Data Corporation (IDC), Tech. Rep., 2009, Document No. 221346. [Online]. Available: www.idc.com

[6] J.-L. Letouzey and T. Coq, "The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code," in Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on, aug. 2010, pp. 43 –48.

[7] ISO/IEC, 9126. Software engineering – Product quality. ISO/IEC, 2001.

[8] ISO/IEC, 25010. Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. ISO/IEC, 2011.

[9] S. Gude and P. Lago, "Best Practices for Energy efficient software," last accessed on February 17, 2012. [Online]. Available: http://wiki.cs.vu.nl/green_software/index.php/Best_practices_for_energy_efficient_software

[10] P. Larsson, "Energy-Efficient Software Guidelines," Intel Software Solutions Group, Tech. Rep., 2011.

[11] G. Kaefer, "Green SW Engineering: Ideas for Including Energy Efficiency into your Software Projects," Online Presentation, Siemens AG, Munich, Germany, 2009. [Online]. Available: http://www.cs.uoregon.edu/events/icse2009/images/postConf/TB-Energy-Efficient-SW-ICSE09.pdf

[12] M. Halstead and R. Bayer, "Algorithm dynamics," in Proceedings of the ACM annual conference, ser. ACM '73. New York, NY, USA: ACM, 1973, pp. 126–135. [Online]. Available: http://doi.acm.org/10.1145/800192.805693

[13] T. McCabe, "A complexity measure," IEEE Trans. Soft. Eng., vol. SE-2, no. 4, pp. 308 – 320, Dec. 1976.

[14] G. McCluskey, "Tuning Java I/O Performance," 1999, last accessed on February 17, 2012. [Online]. Available: http://java.sun.com/developer/technicalArticles/Programming/PerfTuning/

[15] "Junit – a programmer-oriented testing framework for java," last accessed on Feb, 17, 2012. [Online]. Available: http://www.junit.org/

[16] "WattsUp Pro ES," last accessed on Feb, 17, 2012. [Online]. Available: https://www.wattsupmeters.com/secure/products.php?pn=0&wai=0&spec=2

 ISBN: 978-1-61208-259-2